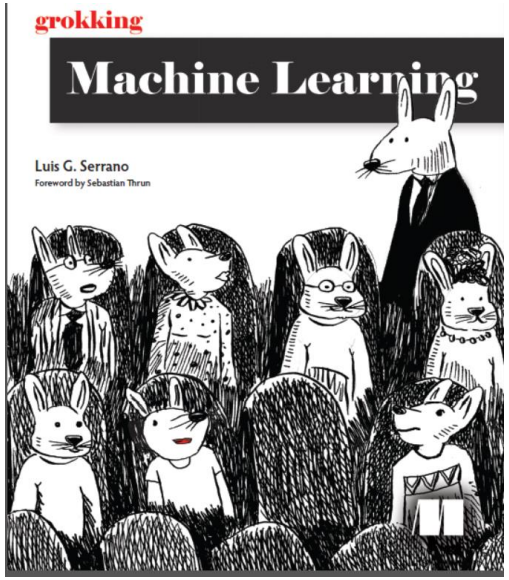


Machine Learning

| Perceptron Algorithm

Prof. Dr. Mostafa Elhosseini

https://youtube.com/@ElhosseiniAcademy

Agenda

- Perceptron Algorithm
- Stopping Criteria
- Linear Separable or Not?
- Batch /Mini-batch / Stochastic Gradient Descent
- Python implementation

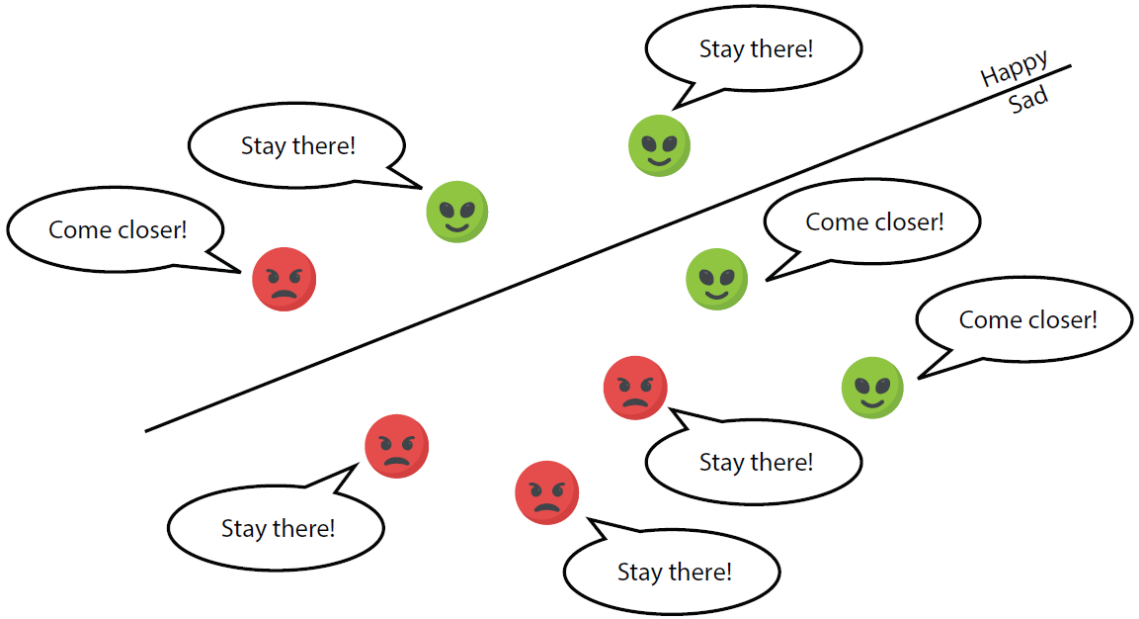
The perceptron trick

Procedure:

- The prediction the perceptron makes at the point is $\hat{y} = step(ax_1 + bx_2 + c)$.
- **Return** the perceptron with the following weights and bias:
  - $a' = a + \eta(y - \hat{y})x_1$
  - $b' = b + \eta(y - \hat{y})x_2$
  - $c' = c + \eta(y - \hat{y})$

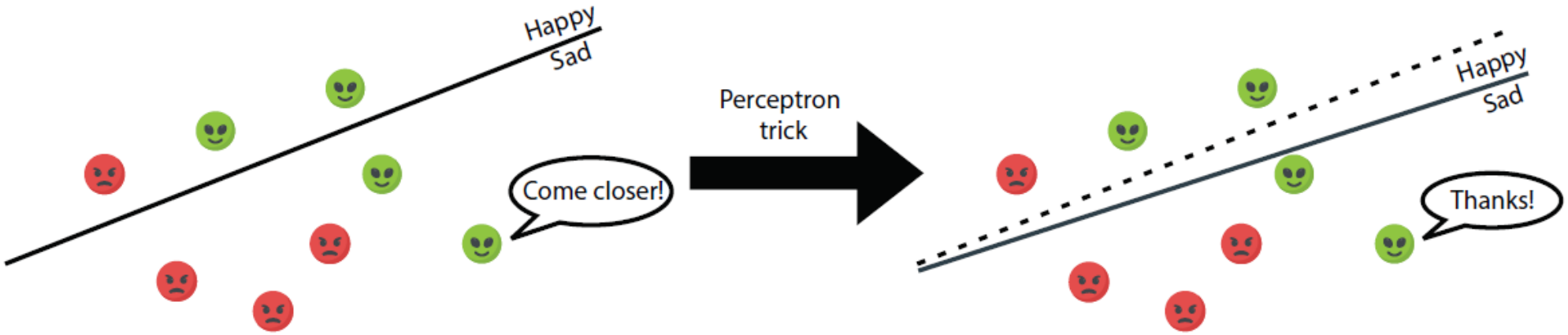
The perceptron Algorithm

- The first step of the algorithm is to **draw a random line**
- It **doesn't** do a good job of splitting the happy and the sad sentences



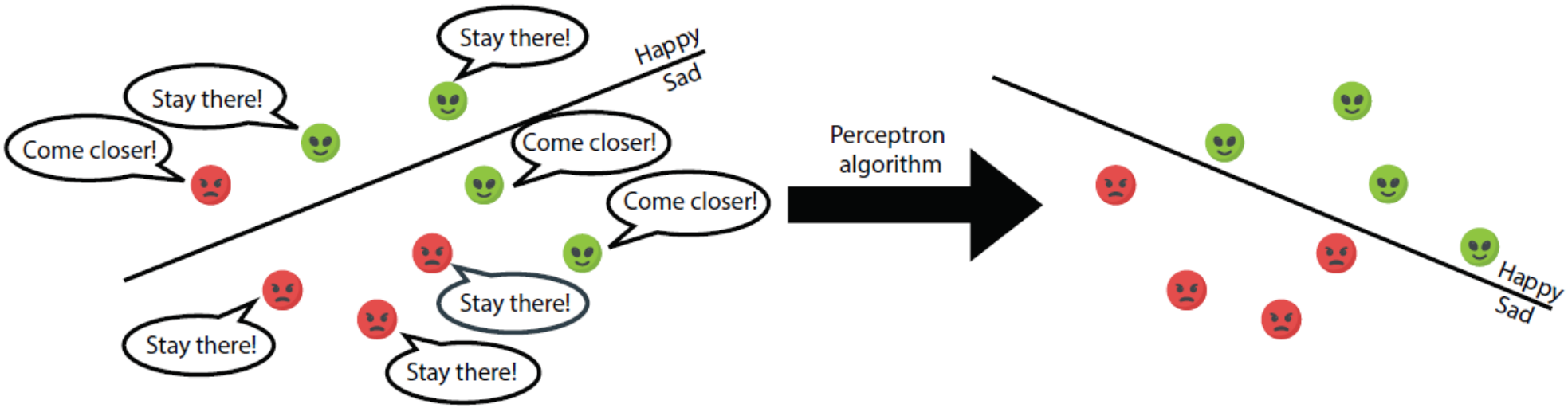
The perceptron Algorithm

- Picking **one point** at random
  - If the point is correctly classified, the line is left alone.
  - If it is misclassified, then the line gets moved slightly closer to the point



The perceptron Algorithm

- Repeat this process many times, eventually we will get to a good solution.
- This procedure doesn't always get us to the best solution.



Hyperparameters

- The algorithm has two hyperparameters: the **number of epochs**, and the **learning rate**

- The number of times we run the algorithm is the number of **epochs**

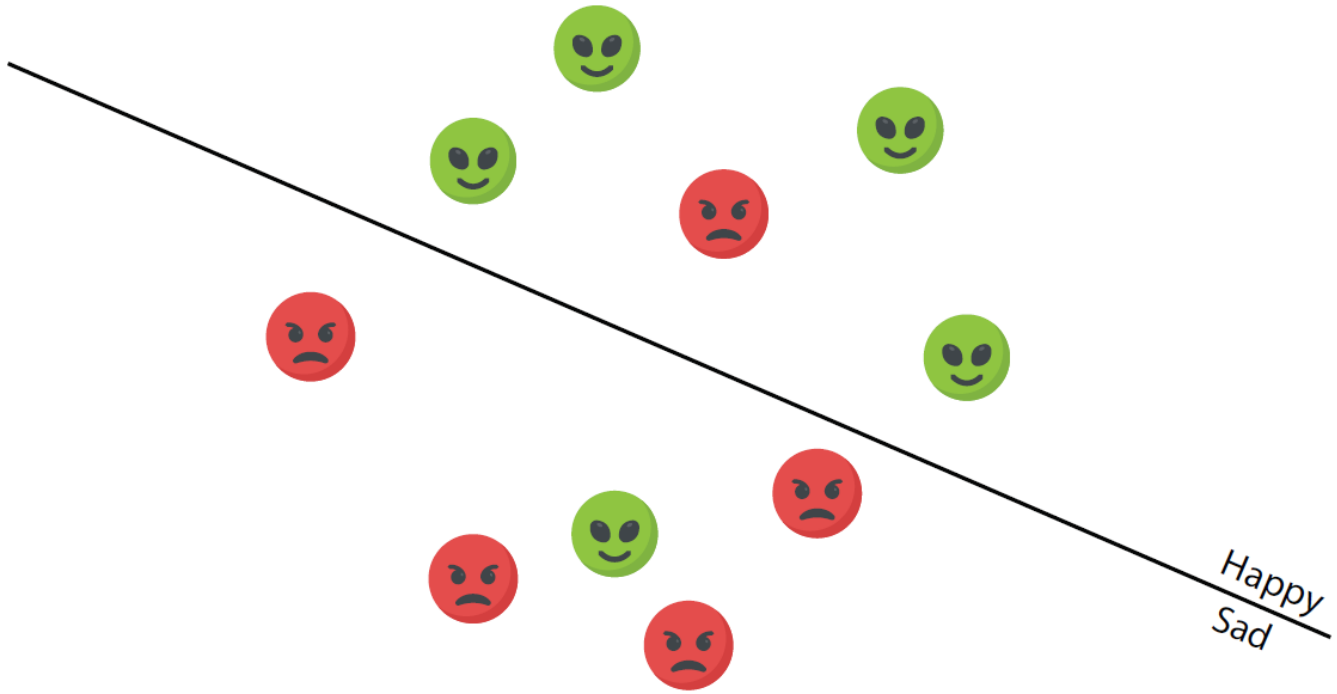
Stopping Criteria

How long should we run the loop? In other words, how many epochs should we use?

- Run the loop a fixed number of times, which could be based on our computing power, or the amount of time we have.
- Run the loop until the error is lower than a certain threshold we set beforehand.
- Run the loop until the error doesn't change significantly for a certain number of epochs.

Not Linearly separable

- It is impossible to find a line to separate the two classes in the dataset



Gradient Descent

- This process uses derivatives

Remind: (Perceptron Trick)

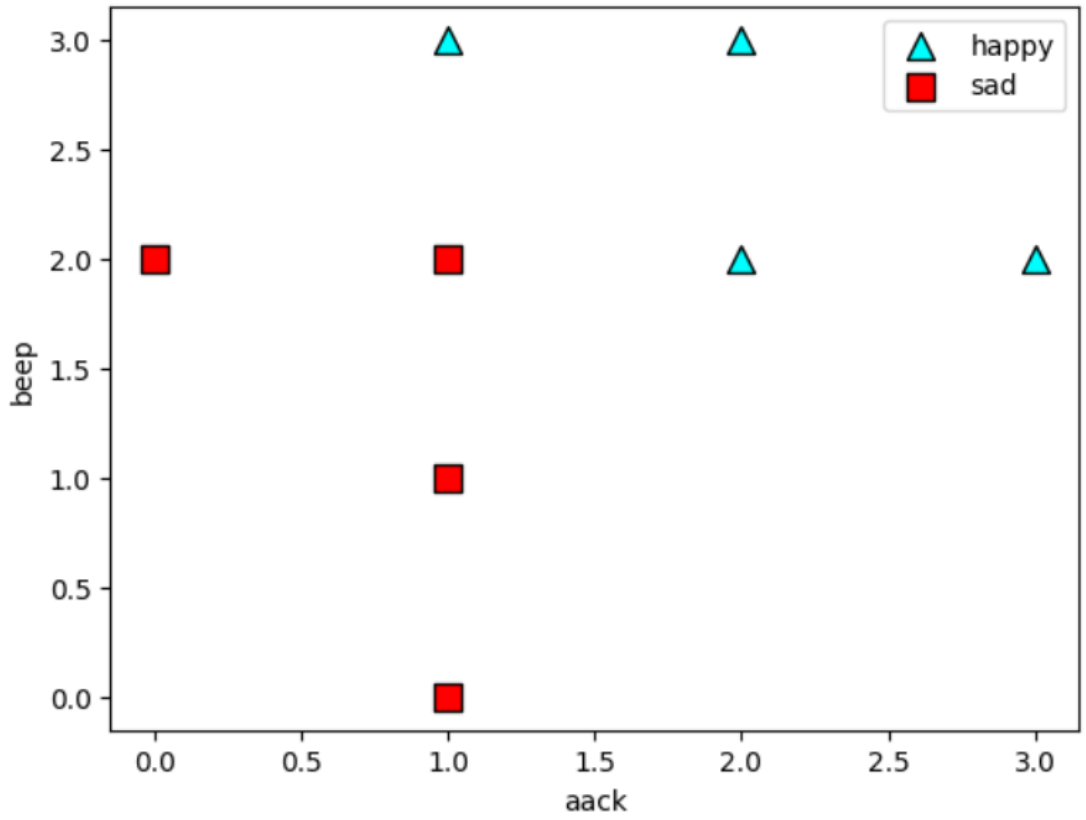
- Every time we "**move a small amount** in this direction," we are calculating in the background a **derivative** of the error function and using it to give us a direction in which to move our line

Stochastic and Batch Gradient Descent

- Repeatedly taking one point at a time and adjusting the perceptron (line) to be a better fit for that point - **Stochastic**
- The better approach is to take a **batch of points at a time** and adjust the perceptron to be a better fit for those points in one step – **Mini Batch**
- The extreme case is to **take all the points** in the set at a time and adjust the perceptron to fit all of them better in one step - **Batch**

Python Session

```
features = np.array([[1,0],[0,2],[1,1],[1,2],[1,3],[2,2],[2,3],[3,2]])
labels = np.array([0,0,0,0,1,1,1,1])
```



Aack	*Beep*	Happy/Sad
1	0	0
0	2	0
1	1	0
1	2	0
1	3	1
2	2	1
2	3	1
3	2	1

Python Session

```
def score(weights, bias, features):
    return features.dot(weights) + bias
```

```
def step(x):
    if x >= 0:
        return 1
    else:
        return 0
```

```
def prediction(weights, bias, features):
    return step(score(weights, bias, features))
```

```
def error(weights, bias, features, label):
    pred = prediction(weights, bias, features)
    if pred == label:
        return 0
    else:
        return np.abs(score(weights, bias, features))
```

```
def mean_perceptron_error(weights, bias, features, labels):
    total_error = 0
    for i in range(len(features)):
        total_error += error(weights, bias, features[i], labels[i])
    return total_error/len(features)
```

Python Session

```
#weights = [1,1]
#bias = -3.5
weights = [1,2]
bias = -4
for i in range(len(features)):
    print(prediction(weights, bias, features[i]), error(weights, bias, features[i], labels[i]))
```

```
0 0
1 0
0 0
1 1
1 0
1 0
1 0
1 0
```

Python Session

```
# First perceptron trick
def perceptron_trick(weights, bias, features, label, learning_rate = 0.01):
    pred = prediction(weights, bias, features)
    if pred == label:
        return weights, bias
    else:
        if label==1 and pred==0:
            for i in range(len(weights)):
                weights[i] += features[i]*learning_rate
            bias += learning_rate
        elif label==0 and pred==1:
            for i in range(len(weights)):
                weights[i] -= features[i]*learning_rate
            bias -= learning_rate
    return weights, bias
```

Python Session

```
# Shorter version of the perceptron trick
def perceptron_trick(weights, bias, features, label, learning_rate = 0.01):
    pred = prediction(weights, bias, features)
    for i in range(len(weights)):
        weights[i] += (label-pred)*features[i]*learning_rate
        bias += (label-pred)*learning_rate
    return weights, bias
```

Python Session

```
perceptron_trick(weights, bias, features[6], 0)
```

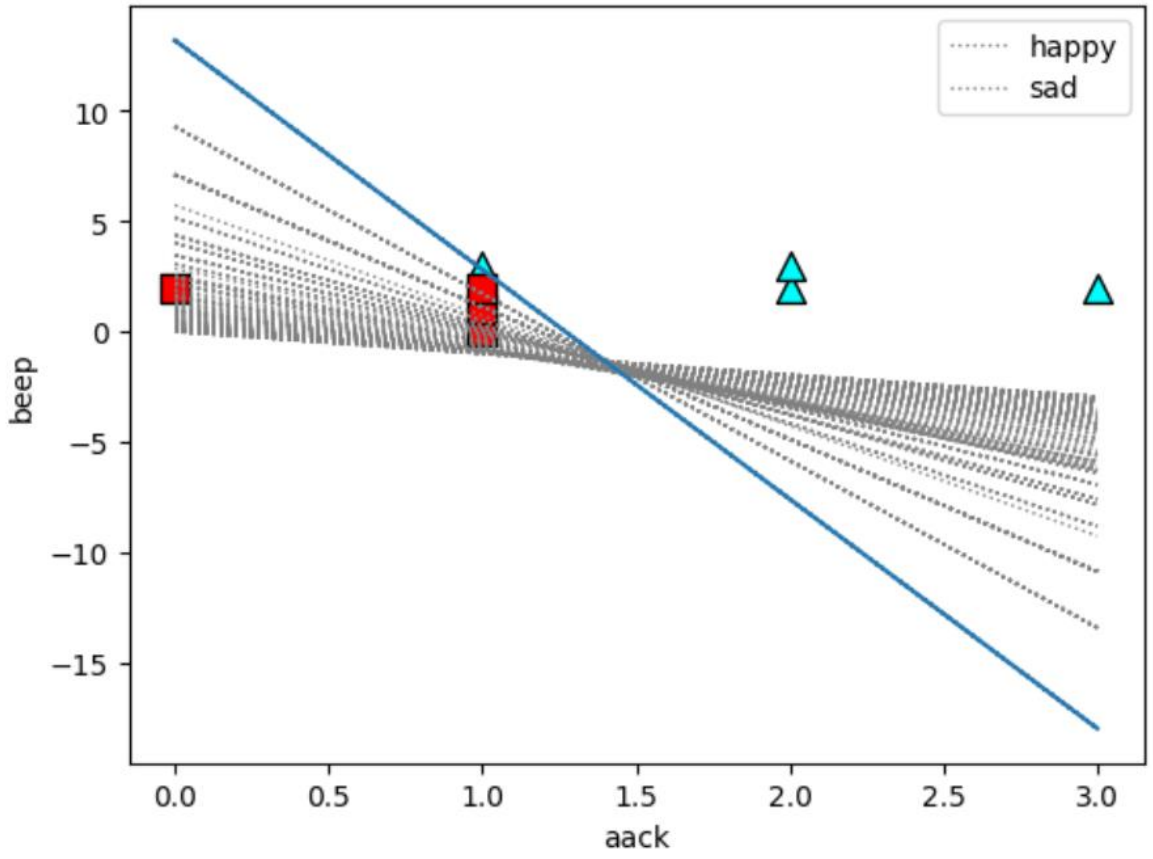
```
([0.98, 1.97], -4.01)
```

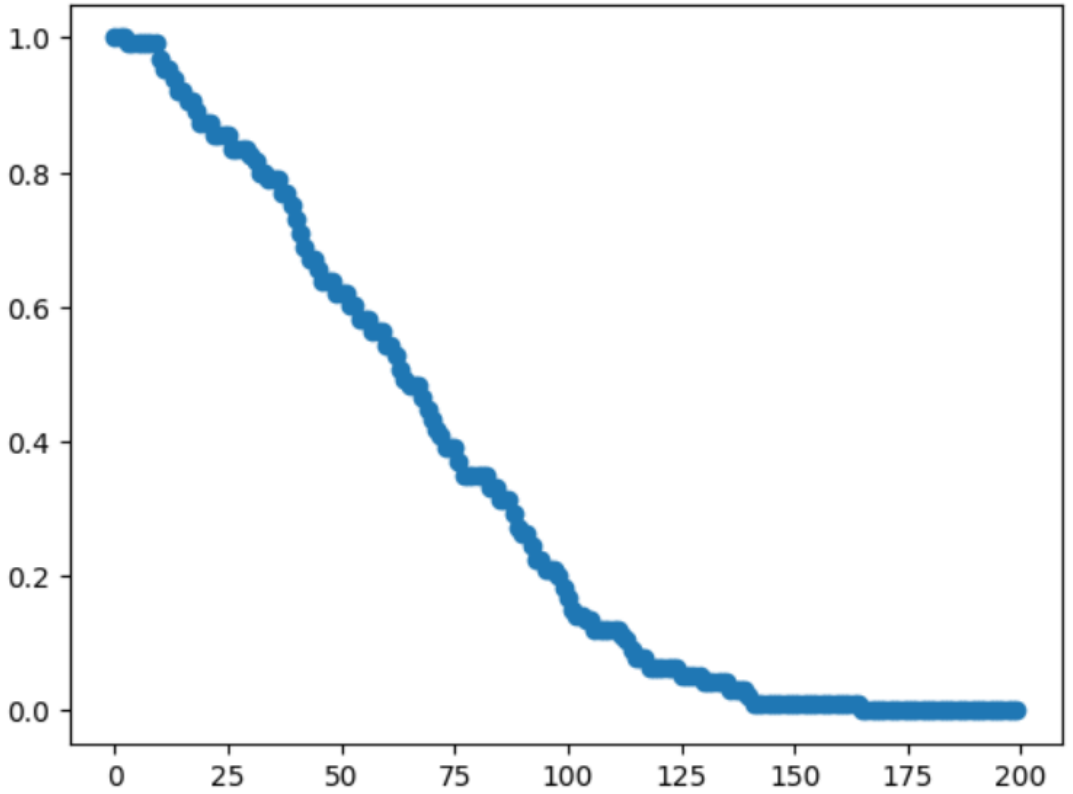
Python Session

```
random.seed(0)
def perceptron_algorithm(features, labels, learning_rate = 0.01, epochs = 200):
    weights = [1.0 for i in range(len(features[0]))]
    bias = 0.0
    errors = []
    for epoch in range(epochs):
        # Coment the following line to draw only the final classifier
        utils.draw_line(weights[0], weights[1], bias, color='grey', linewidth=1.0, linestyle='dotted')
        error = mean_perceptron_error(weights, bias, features, labels)
        errors.append(error)
        i = random.randint(0, len(features)-1)
        weights, bias = perceptron_trick(weights, bias, features[i], labels[i])
    utils.draw_line(weights[0], weights[1], bias)
    utils.plot_points(features, labels)
    plt.show()
    plt.scatter(range(epochs), errors)
    return weights, bias
```

```
perceptron_algorithm(features, labels)
```

Python Session






```
([0.5199999999999996, 0.049999999999999364], -0.6600000000000004)
```

Batch

```python
def perceptron_algorithm_full_batch(features, labels, learning_rate = 0.01, epochs = 200):
    weights = [1.0 for i in range(len(features[0]))]
    bias = 0.0
    errors = []
    for epoch in range(epochs):
        cumulative_weights_update = np.zeros_like(weights)
        cumulative_bias_update = 0.0
        for i in range(len(features)):
            weights_update, bias_update = perceptron_trick(weights, bias, features[i], labels[i], learning_rate)
            cumulative_weights_update += (weights_update - weights)
            cumulative_bias_update += (bias_update - bias)
        weights += cumulative_weights_update / len(features)
        bias += cumulative_bias_update / len(features)
        error = mean_perceptron_error(weights, bias, features, labels)
        errors.append(error)
    return weights, bias
```

Mini batch

```python
def perceptron_algorithm_mini_batch(features, labels, learning_rate = 0.01, epochs = 200, batch_size = 4):
    weights = [1.0 for i in range(len(features[0]))]
    bias = 0.0
    errors = []
    for epoch in range(epochs):
        indices = np.random.permutation(len(features))
        mini_batches = [indices[k:k+batch_size] for k in range(0, len(features), batch_size)]
        for batch in mini_batches:
            cumulative_weights_update = np.zeros_like(weights)
            cumulative_bias_update = 0.0
            for i in batch:
                weights_update, bias_update = perceptron_trick(weights, bias, features[i], labels[i], learning_rate)
                cumulative_weights_update += (weights_update - weights)
                cumulative_bias_update += (bias_update - bias)
            weights += cumulative_weights_update / len(batch)
            bias += cumulative_bias_update / len(batch)
        error = mean_perceptron_error(weights, bias, features, labels)
        errors.append(error)
    return weights, bias
```

Applications

- Spam Filter
- Next Lecture